Towards Finding Accounting Errors in Smart Contracts

Brian Zhang

Blockchains and cryptocurrencies have become an integral part of our economy. As of the writing of this paper, the global market cap for cryptocurrencies reaches $1.19 Trillion USD, with the top two blockchains being Bitcoin and Ethereum. An important kind of blockchain-based applications are smart contracts, which can encompass a wide range of services, from banks to gaming platforms and marketplaces. Smart contracts follow the DeFi, or *decentralized finance* principle. Unlike centralized systems, such as federal banks, smart contracts operate in a decentralized manner without a single controlling authority, rendering many novel financial applications. Similar to traditional software, smart contracts are developed by programmers and inevitably have bugs. The lucrative value of exploiting these bugs has made smart contracts one of the most popular targets of many malicious actors. As of Q2 of 2023, $300 million USD were exploited from 212 security incidents, suggesting that each exploit costed an average of $1.5 million USD.

Therefore, there is a pressing need to develop techniques to find smart contract bugs. Existing techniques can be roughly classified into four categories: static analysis, fuzzers, symbolic execution, and verification. Static analysis toolsanalyze source code without actually running the code. They usually transform smart contracts to various intermediate representations and then search for certain bug patterns. Fuzzers run contracts against a large number of inputs and transactions sequences. Symbolic execution analyzes all possible program paths of a smart contract by performing symbolic computation instead of concrete execution. Verification toolsleverage formal methods to check smart contracts against formal specifications. These approaches have demonstrated great effectiveness in identifying a broad range of issues. Some bugs such as *reentrancy* and integer *overflow* and *underflow* can hardly survive these tools. However, most automatic techniques rely on application agnostic oracles, meaning that bugs need to be clearly defined without considering application specific semantics. Such oracles may be difficult to acquire for certain kinds of bugs. Verification tools are capable of detecting a wide spectrum of bugs including those that are application specific. However, they need the developers to provide application specifications, which may entail substantial manual efforts. As a result, there are still a large number of bugs that are beyond existing tools, evidenced by the growing number of attacks.

According to a recent study by Zhang et al. [1] on over 500 exploitable bugs (bugs that can lead to direct fund loss) from 119 real-world smart contract projects, 80% of exploitable bugs are *machine unauditable bugs* (MUBs), meaning that they fall outside of the scope of existing automatic tools. Among them, *financial bugs*, which are incorrect implementations of underlying contract business models, are the most common type of MUBs in projects before deployment and also the second hardest to find in manual auditing, due to the need of understanding the most complex parts of contracts, namely, the business logics. On the other hand, their impact can be devastating. An example would be the Uranium Finance Exploit. Due to two extra zeros in an interest calculation, the contract was exploited for $57 million USD. The bug survived multiple rounds of manual auditing (by experts).

In this paper, I develop a type-checking tool for accounting bugs in smart contracts. My insight is that *although financial bugs reside in complex business logic and seemingly lack an application-agnostic oracle, many manifest themselves as abstract type violations. Abstract type inference* is a technique that can be traced back to the 70's in the last century. It aims to abstract higher level semantic information such as physical units (e.g., seconds and meters) than those denoted by primitive types in programming languages such as integers and strings. As such, type systems can be enhanced to check a much richer set of properties, such as physical unit consistency in robotic systems. I further observe that *although smart contracts have sophisticated business models, their basic operations are still analogous to those in a simple bank system*, such as deposit, withdraw, exchange, and loan. I hence devise an abstract type system based on these operations that can infer and check abstract types. In particular, I model and infer three facets of each variable, which are: *token unit* indicating the kind of currency denoted by the variable (analogous to USD in real life), *scaling factor* that denotes how much the variable has been scaled in order to simulate floating point computation that is not supported in smart contract programming languages, and *financial meaning*, e.g., if the variable denotes an interest or a debt. With the rich types, I can check a large set of properties that shall be uniformly true for various business models, using type rules. For instance, values of different token units cannot be added or subtracted together, similar to how lengths of meters and inches cannot be added together; amounts scaled by different factors should not be compared; interest should not be subtracted from debt but rather adds to it. To use my tool, the user annotates a few global variables and function parameters with their abstract types. The annotations are limited (see **??**) and usually clear from project description and even variable names. Then, my technique automatically infers the abstract types for other variables and performs type checking.

I implement a prototype ScType based on Slither [2]. I evaluate the system on 50 real-world contracts from [1] and Code4Rena [3]. It finds 31 bugs with 87.9% recall and 73.8% precision. In contrast, the state-of-the-art tools could only find 5 of the bugs. The paper has been accepted to ICSE 2024 (a prestigious research conference in Software Engineering) to be held in Portugal. My system can be downloaded and tested at [4].

REFERENCES

[1] Z. Zhang, B. Zhang, W. Xu, and Z. Lin, "Demystifying exploitable bugs in smart contracts," *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 615–627, 2023.

[2] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *WETSEB@ICSE*. IEEE / ACM, 2019.

[3] "Code4rena." [Online]. Available: https://code4rena.com

[4] "sctype," 2023. [Online]. Available: https://hub.docker.com/repositories/icse24sctype